BURTEAUX Pierre

CHAUDOIN Julien

MATHIEU Rodolphe

NEVES Clément

PAPELIER Romain

POTIER Thomas

Projet de Compilation

CONTENU

Introduction 3

Gestion du projet 3

Répartition du travail 3

Durée de réalisation 4

Méthode de travail 4

Outils utilisés 5

Fonctionnement du compilateur 6

Etat du projet 8

Difficultés rencontrées 9

Conclusion 10

INTRODUCTION

L'objectif de ce projet était de comprendre le fonctionnement d'un compilateur ainsi que d'en créer un fonctionnel . Le projet a pu être réalisé grâce aux cours magistraux, aux travaux pratiques ainsi qu'au travail fourni en dehors de ces derniers :

Pour mener à bien ce projet, les prérequis étaient :

1. Maîtriser les méthodes et les outils de traitement de flots de de caractères : **CUP et LEX**
2. Maîtriser le langage cible : **Assembleur**
3. Savoir créer une grammaire à partir des outils CUP et LEX
4. Savoir gérer une table des symboles permettant de répertorier les variables et les fonctions (ainsi que leur propriétés)
5. Savoir construire un arbre abstrait

GESTION DU PROJET

REPARTITION DU TRAVAIL

Pour réaliser ce projet nous avons combiné séances de travail en groupe et travail individuel. Lorsque tous les membres du groupe étaient présents sur Nancy, nous avons organisé des séance de travail en groupe ; essentiellement pour poser les bases du projets. Ensuite, nous avons utiliser Github qui nous a permis de centraliser le travail de chacun.

Nous nous sommes répartis à chacun des taches bien précises, de manière à ne pas perdre de temps à gérer la gestion du projet.

DUREE DE REALISATION

Durée total :

Nous avons organisé notre première réunion le 29 mars. A partir de cette date, le projet avançait quasiment quotidiennement. Nous l'avons terminé le 26 avril, soit une durée totale d'environ un mois.

Répartition du temps par tâches :

- Grammaire : 30%
- Cup : 20%
- Lex : 5%
- TDS : 20%
- Arbre : 15%
- Générateur : 10%

METHODE DE TRAVAIL

Lors des séances de TP :

Les séances de TP nous ont permis d'avoir une base assez solide pour entamer le projet. Les séances de TP consacrées à la création de grammaire, à la génération de code assembleur et autre nous ont donc été très instructrices.

Durant ces séances nous avons également pu obtenir des réponses sur des points du projet qui nous paraissaient flous (notamment sur le fonctionnement de CUP et LEX), ou avoir des explications sur certaines parties du code (notamment en assembleur).

En dehors des séances de TP :

- Répartition des tâches
- Travail individuel sur les tâches à effectuer
- Travail en commun sur les tâches à effectuer
- Propositions d'idées
- Mise en commun sur un dépôt GitHub

OUTILS UTILISES

Création de la grammaire :

Pour la création de la grammaire, nous avons utilisé deux outils. Nous avons utiliser CUP afin de générer un parser (et ainsi créer la table des symboles ainsi que l'arbre abstrait). Nous avons également LEX, qui est un analyseur lexical.

Outils de développement :

Pour le développement du projet nous avons tous utilisé le même IDE, à savoir Eclipse. Nous avons également eu à utiliser BSIM.

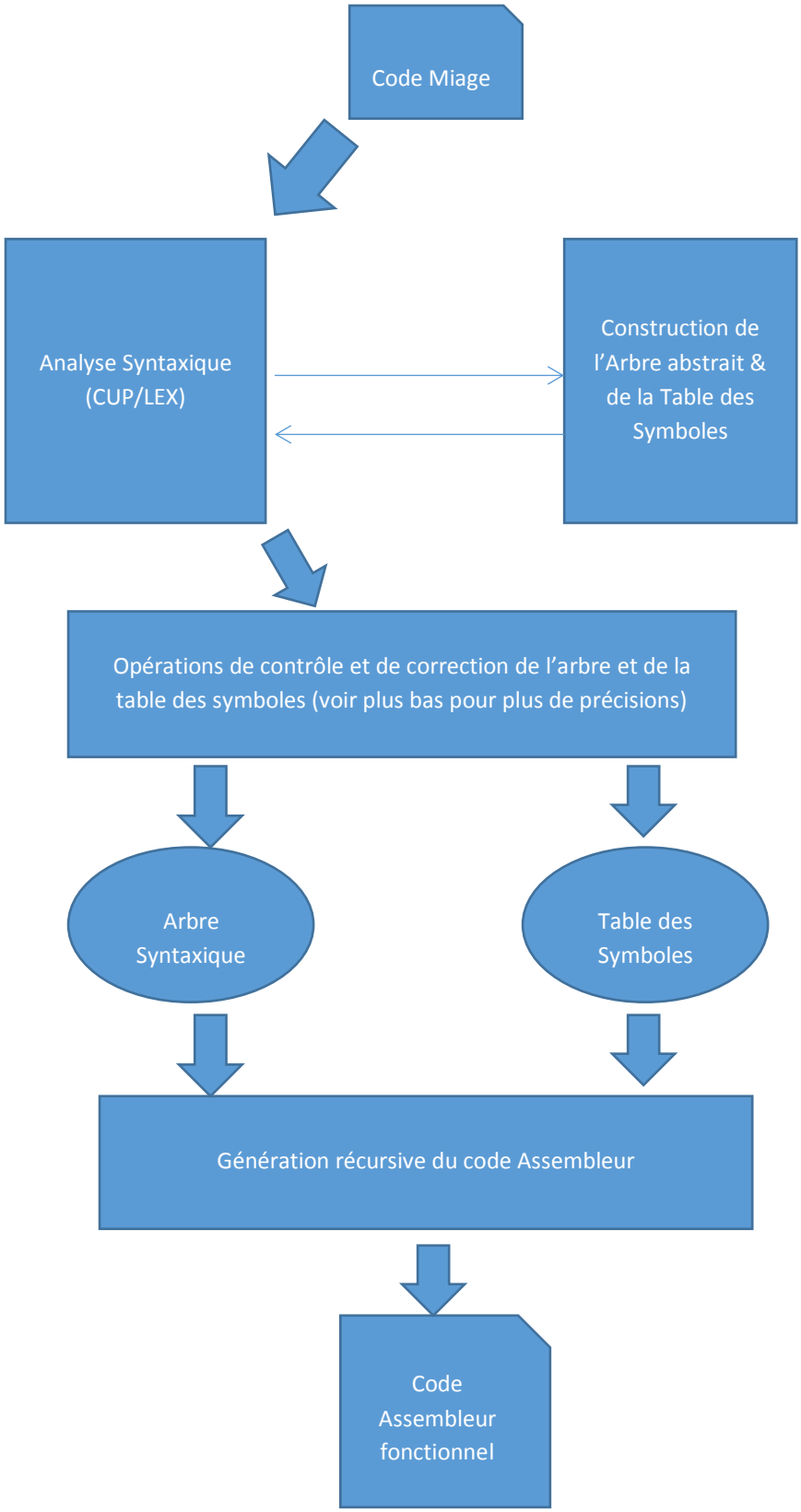
Outils de travail collaboratif :

Lors de la phase de travail individuel nous avons utilisé Github pour centraliser le travail de chacun. Nous l'avons choisit car il est simple d'utilisation et fonctionne sur les toutes les plateformes (Mac OSX, Windows et Linux). Nous avons également utilisé de la messagerie instantané pour communiquer entre nous.

FONCTIONNEMENT DU COMPILATEUR

Notre compilateur fonctionne selon les étapes suivantes :

- Il prend un fichier ".miage" en entrée.
- Il l'analyse syntaxiquement, grâce à notre grammaire CUP et à l'analyseur LEX associé.
- Si le compilateur trouvait une erreur de syntaxe en analysant le fichier, il devait arrêter le traitement, et passer à un autre fichier.
- S'il le fichier était correcte, le compilateur devait créer la table des symboles et l'arbre abstrait du fichier.
- Pour finir, le compilateur devait générer le code assembleur correspondant en sortie, de manière entièrement récursive, en fonction de chaque nœud.


Code Miage
Analyse Syntaxique (CUP/LEX)
Construction de l'Arbre abstrait & de la Table des Symboles
Opérations de contrôle et de correction de l'arbre et de la table des symboles (voir plus bas pour plus de précisions)
Arbre Syntaxique
Table des Symboles
Génération récursive du code Assembleur
Code Assembleur fonctionnel

Les opération de contrôle de l'arbre et de la table des symboles concernent les opération de contrôle sémantique (conflit de type, doubles déclaration, etc) mais concernent également des opérations de correction de l'arbre concernant les identificateurs et leurs contexte. En effet, l'exécution de « l'action code » d'une règle de grammaire se fait une fois cette règle entièrement lue, ce qui veut dire que l'action code d'une règle « fille » sera exécutée avant l'action code de la règle mère. Ce qui nous menait à un problème de gestion de contextes : les variables locales utilisées dans une fonction étaient définie dans la TDS avant que cette fonction ne soit définie dans cette même TDS. Nous avons donc décidé de créer des nœuds temporaires de type « DEF » permettant, à la fin de la construction de l'arbre, de compléter / corriger la table des symboles et définir les bons contextes dans celle-ci. Ces opérations sont effectuées dans l'action code de la règle racine (Nœud « PROG »), car exécutée en dernier.

ETAT DU PROJET

Nous avons implanté toutes les fonctions nécessaire au compilateur pour que celui-ci réalise son travail (*voir section "Fonctionnement du compilateur"*). Tous les jeux de test passent sans problème à l'exécution, excepté le jeu numéro 00, car nous n'avons pas programmé les méthode de génération pour les commandes Read() et Write(). Il reste donc quelques améliorations possibles à réaliser :

- Contrôle sémantique concernant la clause RETURN : le programme java actuel ne vérifie pas la présente d'un RETURN à l'intérieur d'une fonction typée non void.
- Contrôle sémantique concernant la déclaration de fonction imbriquée : le programme java actuel ne vérifie pas si une fonction est déclarée à l'intérieur d'une autre fonction (ce qui n'est pas censé être possible pour notre langage)
- Syntaxe des conditions booléennes : le programme actuel ne gère que des conditions écrites sous la forme "expression1 OPERATEUR_BOOLEEN expression1", où expression1 et expression2 sont des expressions à valeur entière (x+y, f(), 3, ...), et non des expression booléennes.
- Génération de code assembleur pour les commandes read() et write() : actuellement, le générateur d'assembleur ne prend pas en compte ces commandes dans le code miage.

On peut voir qu'il ne reste que très peu de choses à réaliser pour que notre compilateur soit totalement fini et fiable. On peut donc dire que notre compilateur est plutôt complet.

DIFFICULTES RENCONTREES

L'apprentissage des nouveaux outils CUP et LEX fut le premier pas et la première difficulté, notamment sur le fait que les expressions régulières de LEX diffèrent légèrement des expressions régulières utilisées en Shell.

La seconde difficulté fut la prise en main du langage Assembleur, étant donné qu'il s'agit d'un langage bas niveau, et notamment l'écriture des méthodes de génération de code Assembleur à partir de l'arbre abstrait.

Chaque fichier de test .MIAGE a été testé pas à pas sur BSIM, ce qui fut là encore une difficulté dans le sens ou le code assembleur en exécution était très dur à lire puisque que toutes les commandes "avancées" (CMOVE, PUTFRAME, PUSH,…) sont des macros qui utilisent toutes ADDC, ST, LD, etc ; et que ce sont ces macros qui étaient affichées lors de l'exécution sur BSIM. Il était donc difficile de se repérer par rapport au code généré par le compilateur.

Enfin les dernières difficultés rencontrées furent directement liées à BSIM : l'une était la taille insuffisante de la mémoire de BSIM, qui ne disposait que de 500 octets, ce qui provoquait des débordements lorsque les programmes générés étaient trop longs (ex: code original trop long ou trop d'appels récursifs), nous empêchant notamment de tester les codes concernant les itérations, les conditionnelles, et la récursivité. Nous avons alors demandé de l'aide à notre enseignant, qui a pu nous fournir une version de BSIM disposant d'une mémoire sur 64Ko.

La dernière difficulté, qui ne fut pas entièrement résolue, était un problème d'Interrupted Exception lancée par BSIM lorsqu'il rencontrait un HALT() dans un code qui comportait un JMP(), peu importe son emplacement dans le code. Cette exception faisait revenir l'exécution du code assembleur à l'instruction 004, ce qui faisait boucler BSIM indéfiniment. N'ayant aucune idée de ce qui pouvait provoquer cette exception, malgré les recherches sur internet, nous avons décidé de placer une commande HALT() à l'instruction 004 pour stopper le programme lorsque l'exception serait levée.

CONCLUSION

Le projet de compilation était très enrichissant car il nous a permis de pourvoir mettre en application directement les éléments étudiés en cours et ainsi donc, de pouvoir mêler la théorie à la pratique.

Il nous a permis de comprendre le fonctionnement d'un compilateur. Il nous a aussi permis d'apprendre à utiliser de nouveaux outils de programmation tels que CUP et LEX.

La gestion du travail en groupe était très enrichissante. Cela nous a permis de nous préparer à travailler sur des projets de taille conséquente, à plusieurs, en se répartissant les tâches et en se coordonnant entre les membres, tout en respectant une Deadline.